Ra Documentation

Release 1.1.1

RA Systems

Oct 19, 2020

Contents

A light-weight effective framework to create business application equipped with a reporting engine and a responsive dashboard written in Python and built on the Django web framework.

CHAPTER 1

Features

- Default Base models to build on your implementation
- Reporting Engine that filters and compute several types of reports with simple lines of code.
- Charting capabilities to turn reports into attractive charts.
- widget system to display reports and its charts on dashboard home , or on object's *view* pages.
- Tools and goodies to extend and customize the framework behavior from top to bottom.
- Python 3.6 / 3.7 / 3.8
- Django 2.2 / 3.0

1.1 Getting started

1.1.1 Installation

```
$ pip install django-ra-erp
```

1.1.2 Create an empty project form scratch

1. Create a virtual environment and install ra-framework from Pypi

```
$ mkvirtualenv ra-env (or use `virtualenv ra-env` if you don't have mkvirtualenv)

$ pip install django-ra-erp
```

2. Ra provides a command to generate a new project with all needed settings in place.

```
$ ra-admin start project_name
```

This will create a django project under the directory *project_name*.

3. Run the usual commands needed for any django project

```
$ ./manage.py migrate
$ ./manage.py createsuperuser
$ ./manage.py runserver
```

4. Done !! Your site should now up and running at *http://localhost:8000*. Enter your super user credentials and login.

1.1.3 Integrating into an existing Project

Please follow to the next section *Integrating Ra into an existing project*

1.1.4 Running the tests

To run the test suite, first, create and activate a virtual environment. Then clone the repo, install the test requirements and run the tests:

```
$ git clone git+git@github.com:ra-systems/RA.git
$ cd cd ra/tests
$ python -m pip install -e ..
$ python -m pip install -r requirements/py3.txt
$ ./runtests.py
# For Coverage report
$ coverage run --include=../* runtests.py [-k]
$ coverage html
```

For more information about the test suite and contribution, we honor https://docs.djangoproject.com/en/dev/internals/contributing/writing-code/unit-tests/.

Integrating Ra into an existing project

First, install the `django-ra-erp` package from PyPI:

```
$ pip install django-ra-erp
```

and/or add the package to your existing requirements file.

Settings

- In your settings file, add the following apps to `INSTALLED_APPS`:

```
INSTALLED_APPS = {
    # ...

    'jazzmin',
    'crequest',
    'crispy_forms',
    'reversion',
    'tabular_permissions',
    'ra',
```

(continues on next page)

(continued from previous page)

```
    'ra.admin',
    'ra.activity',
    'ra.reporting',
    'sample_erp',
    'slick_reporting',
    'django.contrib.admin', # comes at the end because the theme is replaced

}
```

- Add the following entries to `MIDDLEWARE`:

```
MIDDLEWARE = {
    # ...
    'crequest.middleware.CrequestMiddleware',
}
```

- Add the following entries to `TEMPLATES` `context_processors`

```
TEMPLATES = {
    'context_processors' = [
        #...
        'django.template.context_processors.i18n',
        'django.template.context_processors.static',
        'ra.base.context_processors.global_info',
    ]
}
```

- Add a `STATIC_ROOT` setting, if your project does not have one already:

```
STATIC_ROOT = os.path.join(BASE_DIR, 'static')
```

- Add `MEDIA_ROOT` and `MEDIA_URL` settings, if your project does not have these already:

```
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
MEDIA_URL = '/media/'
```

- Ra uses django-crispy-forms bootstrap 4 for the reporting forms. So we need to add this:

```
CRISPY_TEMPLATE_PACK = 'bootstrap4'
```

- Add default Jazzmin theme Settings

```
JAZZMIN_SETTINGS = {
    'navigation_expanded': False,
    "changeform_format": "single",
}

JAZZMIN_UI_TWEAKS = {
    "navbar": "navbar-primary navbar-dark",
    "no_navbar_border": True,
    "body_small_text": False,
    "navbar_small_text": False,
    "sidebar_nav_small_text": False,
    "accent": "accent-primary",
    "sidebar": "sidebar-dark-primary",
    "brand_colour": "navbar-primary",
```

(continues on next page)

(continued from previous page)

```
    "brand_small_text": False,
    "sidebar_disable_expand": False,
    "sidebar_nav_child_indent": True,
    "sidebar_nav_compact_style": False,
    "sidebar_nav_legacy_style": False,
    "sidebar_nav_flat_style": False,
    "footer_small_text": False
}
```

- Finally, you can add a `RA_SITE_TITLE` - which will be displayed on the main dashboard of the Ra dashboard:

```
RA_SITE_TITLE = 'My Example Site'
```

Various other settings are available to configure Ra's behaviour - see *Settings*.

URLS configuration

We need to hook the dashboard / Ra admin site in `urls.py`, like so:

```
from django.urls import path
from ra.admin.admin import ra_admin_site

urlpatterns = [
    # ...
    path('your-url-here', ra_admin_site.urls),
    # ...
]
```

With this configuration in place, you are ready to run `./manage.py migrate` to create the database tables used by Ra.

User accounts

Superuser accounts receive automatic access to the Ra Dashboard interface; use `./manage.py createsuperuser` if you don't already have one.

Start developing

You're now ready to add a new app to your Django project via `./manage.py startapp`.

Follow to the tutorial to create sample erp system which tracks sales and expense and profitability. *Sample ERP System Tutorial*

System Design / Components

Components

1. **Base Models** There are base abstract models which holds attributes and method that the system expect. Those attributes and fields are very generic. For reference: *Base Classes*
2. **Admin Site and Models** A custom admin site and admin.ModelAdmin subclasses that holds functionality needed for a smooth framework dashboard experience For reference: *The Admin*

3. **Report Registry and Views** A Registry for reports created. It's also responsible for report menu generation. *ReportView* is a subclass of slick_reporting.ReportView with many addition like caching and ajax.
4. **Front End Report/Widget Loader** A collection of Javascript / jQuery function and wrappers to easily create chart/report widgets and take full control on how they are displayed.

Why using the admin?

It's much simpler especially around CRUD intensive apps (like ERPs). * With just one class you can manage all CRUD operations. * Easily have your url named and reversed * Multiple Formsets support out of the box. * It's unified, no need to learn new terminologies. If you worked with the admin you'll know your way around here. * Admin goodies (list filter, auto form generation, save and continue, save and add new buttons, discovery of related objects that would get deleted) * Out of the box permissions handling

Now, imagine having to write all of this, again, in class based views.. that's a pain that no one should face.

"But the admin should be for site admins only."

Well, that's an old phrase that gets passed around which i dont find convincing enough. Maybe it was true in the old days; But now, you dont have to be a *staff* member to be able to log in an admin dashboard. Also, Ra dashboard is a custom admin site (independent from your typical admin).

Reporting

Reporting engine itself was moved from this package to be an independent package Django Slick Reporting

Ra framework, away from the calculation itself, holds functionality of organizing the reports and create html widgets out of those reports, which can be controlled . and by default support showing results in tables, and different kinds of charts, all in speed.

1.2 Sample ERP System Tutorial

In this section we will create a sample erp project using Ra framework.

1.2.1 Sample ERP models and admin

Let's use the project we just generated in the Quickstart section and build an app that manages a business, records and report its product sales, as well as its expenses, and finally its profitability .

First we need to create an app

```
$ django-admin startapp sample_erp
```

then add *sample_erp* to your `INSTALLED_APPS`.

Models

To manage a business we would need to track the sales , the clients and the expenses. Here is a sample implementation

```
from django.db import models
from ra.base.models import EntityModel, TransactionModel, TransactionItemModel,␣
↪QuantitativeTransactionItemModel
from ra.base.registry import register_doc_type
from django.utils.translation import ugettext_lazy as _


class Product(EntityModel):
    class Meta:
        verbose_name = _('Product')
        verbose_name_plural = _('Products')

class Client(EntityModel):
    class Meta:
        verbose_name = _('Client')
        verbose_name_plural = _('Clients')


class Expense(EntityModel):
    class Meta:
        verbose_name = _('Expense')
        verbose_name_plural = _('Expenses')


class ExpenseTransaction(TransactionItemModel):
    expense = models.ForeignKey(Expense, on_delete=models.CASCADE)

    class Meta:
        verbose_name = _('Expense Transaction')
        verbose_name_plural = _('Expense Transactions')


class SalesTransaction(TransactionModel):
    client = models.ForeignKey(Client, on_delete=models.CASCADE)

    class Meta:
        verbose_name = _('Sale')
        verbose_name_plural = _('Sales')


class SalesLineTransaction(QuantitativeTransactionItemModel):
    sales_transaction = models.ForeignKey(SalesTransaction, on_delete=models.CASCADE)
    product = models.ForeignKey(Product, on_delete=models.CASCADE)
    client = models.ForeignKey(Client, on_delete=models.CASCADE)

    class Meta:
        verbose_name = _('Sale Transaction Line')
        verbose_name_plural = _('Sale Transaction Lines')
```

The Base Classes we inherit from are fairly straight forward, basically they encapsulate some typical needed fields and offer related method that serves the system integrity.

Example:

- On `EntityModel` those extra fields are *slug*, *notes*, creator user and creation date, and last modified user and last modified date.
- On `TransactionItemModel` extra fields are *value* + the above

- On `QuantitativeTransactionItemModel` extra fields are *quantity*, *price* and *discount* + all the above

You can read more in *Base Classes*, but for now, that pretty much what we need to know.

Run `python manage.py makemigrations sample_erp`, `python manage.py migrate` to update the database with your models

The Admin

Ra makes use of the django admin to leverage the process of authentication, authorization and CRUD operation(s). This is done by

1. Using a different admin site.
2. Using subclasses of ModelAdmin which offer more enhancements.

With this information in mind, let's add the below piece of code into *admin.py*

```
from .models import Client, Product, Expense, ExpenseTransaction,␣
↪SalesLineTransaction, SalesTransaction
from ra.admin.admin import ra_admin_site, EntityAdmin, TransactionAdmin,␣
↪TransactionItemAdmin


class ExpenseAdmin(EntityAdmin):
    pass


class ProductAdmin(EntityAdmin):
    pass


class ClientAdmin(EntityAdmin):
    pass


class SalesLineAdmin(TransactionItemAdmin):
    fields = ('product', 'price', 'quantity', 'value')
    model = SalesLineTransaction


class SalesOrderAdmin(TransactionAdmin):
    inlines = [SalesLineAdmin]
    fields = ['slug', 'doc_date', 'client', ]
    copy_to_formset = ['client']


ra_admin_site.register(Client, ClientAdmin)
ra_admin_site.register(Product, ProductAdmin)
ra_admin_site.register(Expense, ExpenseAdmin)
ra_admin_site.register(SalesTransaction, SalesOrderAdmin)
```

Like with models, here we inherit our admin models from `EntityAdmin`, `TransactionAdmin``and ``TransactionItemAdmin` Also we register our model with their AdminModel with `ra_admin_site` which is an independent admin site than the default django one.

Note: *EntityAdmin* and `TransactionAdmin` are just subclasses of *admin.ModelAdmin*. *TransactionItemAdmin* is a subclass of *admin.TabularInline*. You can customize it as you'd do normally with any ModelAdmin. You can add list_filter(s), select_related, adjust fields and fieldsets on the change_form, etc..

Read more about Admin options: *The Admin*

Let's run and access our Dashboard, enter your username and password created with *createsuperuser*. In the left hand menu you'd find a menu, which will contains links to Clients, Products & SimpleSales admin pages as you'd expect.

Go to the sales order page, add a couple of sale transaction entries. Now, we notice that

1. *value field* is editable, while it should be readonly
2. The Value field should automatically equals the result of price * quantity.

Front End customization

Let's enhance our Sales Page and make *value* a read only

```
from django import forms

class SalesOrderAdmin(TransactionAdmin):
    # ...
    add_form_template = change_form_template = 'sample_erp/admin/sales_change_form.
↪html'

    def formfield_for_dbfield(self, db_field, request, **kwargs):
        formfield = super().formfield_for_dbfield(db_field, request, **kwargs)
        if db_field.name == 'value':
            formfield.widget = forms.TextInput(attrs={'readonly': 'readonly'})
        return formfield
```

Now we need and compute the value automatically and display it to the user. To do that we need to add a little javascript to handle the client side calculation, and to do that we'll need a create our own template.

in you In your *sample_erp* app directory, create a *templates* folder, and inside it you can create a template file *admin/sales_change_form.html* and in it we can write:

```
{% extends 'ra/change_form.html' %}

{% block extrajs %}
    {{ block.super }}
    <script>
            django.jQuery(document).ready(function () {
                const allQuantity = $('[name*=quantity]');
                const allPrice = $('[name*=price]');

                function calculateTotal(e) {
                    let holder = $(e.target).parents('.dynamic-saleslinetransaction_
↪set');
                    let $quantity = holder.find('[name*=quantity]');
                    let $price = holder.find('[name*=price]');
                    let quantity = $.ra.smartParseFloat($quantity.val());
                    let price = $.ra.smartParseFloat($price.val());
                    holder.find('[name*=value]').val(quantity * price)
                }

                allQuantity.on('change', calculateTotal);
                allPrice.on('change', calculateTotal);
```

(continues on next page)

(continued from previous page)

```
                // The newly created rows
                // ref: https://docs.djangoproject.com/en/2.2/ref/contrib/admin/
↪javascript/
                django.jQuery(document).on('formset:added', function (event, $row,␣
↪formsetName) {
                    $row.find('[name*=quantity]').on('change', calculateTotal)
                    $row.find('[name*=price]').on('change', calculateTotal)
                });
            })
    </script>
{% endblock %}
```

Notice here:

1. we *extends* from *ra/change_form.html'* This enables us to change themes of your Ra dashboard rather easily. You can read more about *Theming*
2. we use `$.ra.smartParseFloat()` in the javascript. This is a custom convenience function to handle strings or empty value when numbers are expected (in which case *value* result would be *NaN*. If you want to try just replace smartParseFloat with normal *parseFloat* and enter a string or make empty the quantity and/or price field.

   For list of javascript tools available *Javascript*

Now runserver, go to Sales Order and check the outcome, experiment around.

Next Section we will create interesting reports about product sales, which product being bought by which clients and client total sales.

1.2.2 Reporting and Charting

Before we begin, charts and reporting get more fun and interesting the more data available. So below is a custom management command code that you can use to generate data for the whole current year. This will definitely enhance our experience with this section. Also you can use it to benchmarking Ra Performance.

Generating test data

Create and add below code to 'sample_erp/management/commands/generate_data.py'

```
import random
import datetime
import pytz
from django.core.management import BaseCommand


class Command(BaseCommand):
    help = 'Generates data for simple sales app'

    def add_arguments(self, parser):
        parser.add_argument('--clients', type=int, action='store', help='Number of␣
↪client to get generated, default 10')
        parser.add_argument('--product', type=int, action='store',
                            help='Number of products t0o get generated, default 10')
        parser.add_argument('--expenses', type=int, action='store',
                            help='Number of Expense to get generated, default 10')
```

(continues on next page)

(continued from previous page)

```
        parser.add_argument('--expense-transaction', type=int, action='store',
                            help='Number of records per day,  default 10')

    def handle(self, *args, **options):
        from ...models import Client, Product, SalesTransaction, SalesLineTransaction,
↪ Expense, ExpenseTransaction
        from django.contrib.auth.models import User
        user_id = User.objects.first().pk
        client_count = options.get('clients', 10) or 10
        product_count = options.get('products', 10) or 10
        records_per_day = options.get('records', 10) or 10

        expense_count = options.get('expenses', 10) or 10
        etransaction_per_day = options.get('expense-transaction', 3) or 3

        # Generating clients
        already_recorded = Client.objects.all().count()
        clients_needed = client_count - already_recorded
        if clients_needed > 0:
            for index in range(already_recorded, already_recorded + clients_needed):
                Client.objects.create(title=f'Client {index}', lastmod_user_id=user_
↪id)
            self.stdout.write(f'{clients_needed} client(s) created')

        # Product
        already_recorded = Product.objects.all().count()
        product_needed = product_count - already_recorded
        if product_needed > 0:
            for index in range(already_recorded, already_recorded + product_needed):
                Product.objects.create(title=f'Product {index}', lastmod_user_id=user_
↪id)
            self.stdout.write(f'{product_needed} product(s) created')

        already_recorded = Expense.objects.all().count()
        Expenses_needed = expense_count - already_recorded
        if Expenses_needed > 0:
            for index in range(already_recorded, already_recorded + Expenses_needed):
                Expense.objects.create(title=f'Expense {index}', lastmod_user_id=user_
↪id)
            self.stdout.write(f'{Expenses_needed} Expense(s) created')

        # generating sales
        # we will generate 10 records per day for teh whole current year
        sdate = datetime.datetime(datetime.date.today().year, 1, 1)
        edate = datetime.datetime(datetime.date.today().year, 12, 31)

        client_ids = Client.objects.values_list('pk', flat=True)
        product_ids = Product.objects.values_list('pk', flat=True)
        expense_ids = Expense.objects.values_list('pk', flat=True)

        delta = edate - sdate  # as timedelta
        for i in range(delta.days + 1):
            day = sdate + datetime.timedelta(days=i)
            day = pytz.utc.localize(day)
            for z in range(1, records_per_day):
                chosen_client = random.choice(client_ids)
```

(continues on next page)

(continued from previous page)

```
                SalesLineTransaction.objects.create(
                    doc_date=day,
                    sales_transaction=SalesTransaction.objects.create(doc_date=day,␣
↪client_id=chosen_client,
                                                                        lastmod_user_
↪id=user_id),
                    product_id=random.choice(product_ids),
                    client_id=chosen_client,
                    quantity=random.randrange(1, 10),
                    price=random.randrange(1, 10),
                    lastmod_user_id=user_id
                )

            for z in range(1, etransaction_per_day):
                ExpenseTransaction.objects.create(
                    doc_date=day,
                    expense_id=random.choice(expense_ids),
                    value=random.randrange(1, 10),
                    lastmod_user_id=user_id
                )
            self.stdout.write(f'{day} Done')
            self.stdout.flush()

        self.stdout.write('----')
        self.stdout.write('Done')
```

Then let's run the command

```
$ python manage.py generate_data

# and here with the default arguments in case you want to fine tune
$ python manage.py generate_data --clients 10 --products 10 --records 10 --expense 10␣
↪--expense-transaction 3
```

Now we have some test data to give us a more complete look. Let's create some reports!!

1.2.3 Creating Reports

In our *sample_erp* app, let's create a *reports.py* file *it can be any name, this is just a convention*. in this file we will be creating our report classes

How much each Client bought (in value)

Below code in a sample report class structure to answer this question

Add it to reports.py

```
from django.utils.translation import ugettext_lazy as _
from ra.reporting.registry import register_report_view
from ra.reporting.views import ReportView
from .models import Client, SalesLineTransaction, Product


@register_report_view
```

(continues on next page)

(continued from previous page)

```
class ClientTotalBalance(ReportView):
    report_title = _('Clients Balances')

    base_model = Client
    report_model = SalesLineTransaction

    group_by = 'client'
    columns = ['slug', 'title', '__balance__']
```

Now, we need to load *reports.py* during the app life cycle so our code is executed. Best way to do such action is in AppConfig.ready

```
# in sample_erp __init__.py
default_app_config = 'sample_erp.apps.SampleERPConfig'

# in sample_erp/apps.py
from django.apps import AppConfig


class SampleErpConfig(AppConfig):
    name = 'sample_erp'

    def ready(self):
        super().ready()
        from . import reports
```

Now re-run *runserver*, go to to the dashboard, You'll find a new menu **Reports** which would contains a *Client* sub menu. Click on the Clients menu will open the Client Report List, which will load the first report automatically.

We can notice that

1. Report table is sortable and searchable (Thanks to datatables.net)
2. Report can also be exported to Excel, can also be printed with a dedicated html template
3. You can filter by *Date* , *Client* and *Product*. For the later two, the widget allow you to select multiple objects.
4. All filters and calculation are done automatically.

Let's create another report that answers the following question

How much each product was sold?

```
@register_report_view
class ProductTotalSales(ReportView):
    # Title will be displayed on menus, on page header etc...
    report_title = _('Product Sales')

    # What model is this report about
    base_model = Product

    # What model hold the data that we want to compute.
    report_model = SalesLineTransaction

    # The meat and potato of the report.
    # We group the records in SimpleSales by Client ,
    # And we display the columns `slug` and `title` (relative to the `base_model`␣
↪defined above)
```

(continues on next page)

(continued from previous page)

```
    # the magic field `__balance__` computes the balance (of the base model)
    group_by = 'product'
    columns = ['slug', 'title', '__balance_quantity__']
```

Did you notice that both class definition are almost the same. Main differences are the *base_model* and in *group_by* and we used *__balance_quantity__* which summarize the field "quantity" instead of the field "value".

For more information about available options checkout the Django Slick Reporting documentation Here

Now let's create a 3rd report.

A Client Detailed statement.

Which is a simple list of the sales transaction

```
@register_report_view
class ClientDetailedStatement(ReportView):
    report_title = _('client Statement')
    base_model = Client
    report_model = SalesLineTransaction


    columns = ['slug', 'doc_date', 'doc_type', 'product__title', 'quantity', 'price',
↪'value']
```

Adding Charts

To add charts to a report, we'd need to add to `chart_settings` . Here is an example we will add two charts to teh first report we created *ClientTotalBalance*

```
class ClientTotalBalances(ReportView):
    ...
    chart_settings = [
        {
            'id': 'pie_chart',
            'type': 'pie',
            'title': _('Client Balances'),
            'data_source': ['__balance__'],
            'title_source': 'title',
        },
        {
            'id': 'bar_chart',
            'type': 'bar',
            'title': _('Client Balances [Bar]'),
            'data_source': ['__balance__'],
            'title_source': 'title',
        },
    ]
```

Reload your development server and check how those charts are displayed in the Client Balances report.

Neat right ?

So to create a report we need to a dictionary to a `chart_settings` list containing

- id: (optional) Name used to refer to this exact chart in front end (we will use that in *Adding charts & report widgets*) default is *type-{index}*
- type: what kind of chart it is bar, pie, line, column
- data_source: a list of Field name(s) of containing the numbers we want to chart,
- title_source: a list label(s) respective to the *data_source*
- title: the chart title

Time Series

A time series is a report where the columns represents time unit (year/month/week/day)

Let's see an example

```
@register_report_view
class ProductSalesMonthly(ReportView):
    report_title = _('Product Sales Monthly')

    base_model = Product
    report_model = SalesLineTransaction

    group_by ='product'
    columns = ['slug', 'title']

        # how we made the report a time series report
    time_series_pattern = 'monthly'
    time_series_columns = ['__balance__']
```

Reload your development server , go to Product reports, and check the Product Sales Monthly report.

All we did was adding

- `time_series_pattern` which describe which pattern you want to compute (daily/monthly/yearly)
- `time_series_columns` where we indicated which field to compute for each time series period.

Noticed that `time_series_columns` is a list? This means that we can have more fields computed fpr each period.

In the above report, we computed the sum of *value* of sales for each product, for each period. We can also know the sum of *quantity* of each product for each period as well. Just add `'__balance_quantity__'` to the `time_series_columns` list.

Reload your app and check the results. You should see that for each month, we have 2 fields "Balance QTY" and "Balance"

Now let's add some charts, shall we ?

```
# Add chart settings to your ProductSalesMonthlySeries
@register_report_view
class ProductSalesMonthly(ReportView):
    ...
    chart_settings = [
        {
            'id': 'movement_column_ns',
            'title': _('comparison - Column'),
            'data_source': ['__balance__'],
            'title_source': ['title'],
            'type': 'column',
```

(continues on next page)

(continued from previous page)

```
        },
        {
            'id': 'movement_bar',
            'title': _('comparison - Column - Stacked'),
            'data_source': ['__balance__'],
            'title_source': ['title'],
            'type': 'column',
            # 'stacked': True,
            'stacking': 'normal',
        },
        {
            'id': 'movement_column_total',
            'title': _('comparison - Column - Total'),
            'data_source': ['__balance__', '__balance_quantity__'],
            'title_source': ['title'],
            'type': 'column',
            'plot_total': True,
        },
        {
            'id': 'movement_line',
            'title': _('comparison - line'),
            'data_source': ['__balance__'],
            'title_source': ['title'],
            'type': 'line',
        },
        {
            'id': 'movement_line_stacked',
            'title': _('comparison - Area - Stacked-Percent'),
            'data_source': ['__balance__'],
            'title_source': ['title'],
            'type': 'area',
            'stacking': 'percent',
        },
        {
            'id': 'movement_line_total',
            'title': _('comparison - line - Total'),
            'data_source': ['__balance__'],
            'title_source': ['title'],
            'type': 'line',
            'plot_total': True,
        },
    ]
```

6 charts to highlight the patterns. Reload the development server and *reload the report page* and check the output.

The charts brings our attention that the slops are always rising . . . that's because we're using the `__balance__` report field. which is a *compound* total of the sales. In fact, in those reports, we might be more interested in the *non compound* total, and there is a report field for that which comes by default called `__total__`

Let's change `__balance__` to `__total__` in *ProductSalesMonthly* and check the results for yourself how is it different.

Exercise: I'm confident you can now create a time series report for the Client sales per month, Yeah ?

It would look like something like this

```
@register_report_view
class ClientSalesMonthlySeries(ReportView):
```

(continues on next page)

(continued from previous page)

```
    report_title = _('Client Sales Monthly')

    base_model = Client
    report_model = SalesLineTransaction


    group_by = 'client'
    columns = ['slug', 'title']

    time_series_pattern = 'monthly'
    time_series_columns = ['__total__']
```

You can add charts to this report too !

Cross-tab report

A cross tab report is when the column represents another different named data object

```
@register_report_view
class ProductClientSalesCrosstab(ReportView):
    base_model = Product
    report_model = SalesLineTransaction
    report_title = _('Product Client sales Cross-tab')

    group_by = 'product'
    columns = ['slug', 'title']

    # cross tab settings
    crosstab_model = 'client'
    crosstab_columns = ['__total__']

    chart_settings = [
        {
            'type': 'column',
            'data_source': ['__total__'],
            'plot_total': False,
            'title_source': 'title',
            'title': _('Detailed Columns'),

        },
        {
            'type': 'column',
            'data_source': ['__total__'],
            'plot_total': False,
            'title_source': 'title',
            'stacking': 'normal',
            'title': _('Stacked Columns'),

        },
        {
            'type': 'pie',
            'data_source': ['__total__'],
            'plot_total': True,
            'title_source': 'title',
            'title': _('Total Pie'),
```

(continues on next page)

(continued from previous page)

```
        }
    ]
```

Lke with the time series pattern, we added

- `crosstab_model`: the field representing the model to use as comparison column.
- `crosstab_columns` the report field(s) we want to compare upon, in the crosstab .
- we used `__total__` report field.

1.2.4 Adding charts & report widgets

First let's recap what we did so far

1. we explored the our base models and their respective ModelAdmins
2. we explored how to create several kind reports and how to add multiple charts to a report.

Now let's continue.

Customizing the Home page

Our home page seems little empty, it would be nice if we can have a *report widgets* on it, some tables with relevant data and charts, Yeah ? Let's display total client sales report table directly into our home.

A Full Report Widget

First let's create a template `sample_erp/index.html` template, then in our settings.py we set this template to be displayed as the index page

```
RA_ADMIN_INDEX_PAGE = 'sample_erp/index.html'
```

And in the our template we add code like this

```
{% extends 'ra/base_site.html' %}
{% load ra_admin_tags ra_tags %}

{% block content %}
    <div class="col-sm-12">
        {% get_report base_model='product' report_slug='ProductSalesMonthly' as␣
↪ProductSalesMonthly %}
        {% get_html_panel ProductSalesMonthly %}
    </div>
{% endblock %}
```

Then let's visit our home page, you should see the charts and the report table of the ProductSalesMonthly report.

`get_html_panel` is a shortcut. Let's add another report without using it to see how it looks

```
    <div class="col-sm-12">
    {% get_report base_model='client' report_slug='ClientTotalBalance' as client_
↪balances %}
    <h2>{{ client_balances.report_title }}</h2>
```

(continues on next page)

(continued from previous page)

```
    <div data-report-widget
         data-report-url="{{ client_balances.get_url }}">

        // include this to load a report chart
        <div data-report-chart style="width:100%; height:400px;"></div>

        // include this to load the report table
        <div data-report-table>

        </div>
    </div>

</div>
```

The above code loaded the first chart as default. If you want to change the chart to another one available, just add attribute `data-report-default-chart="YOUR_CHART_ID"` to the `[data-report-chart]` element

Customizing the View page

Ra also provide a view page for each EntityModel subclass, registered with *EntityAdmin*. For example: If you go to the Clients change list page, you'd find a column called "Stats" which will redirect you to a blank page with the title *Statistics for <Client name>*

Same like what we did with the home page, we can add widgets to be displayed for this specific object. Let's see how.

First we need a custom template, so lets create *sample_erp/admin/client_view.html* and assign it to the model admin *view_template*

in *sample_erp/admin.py*

```
class ClientAdmin(EntityAdmin):
    ...
    view_template = 'sample_erp/admin/client_view.html'
```

And in *sample_erp/admin/client_view.html* let's reuse the exact same code we used in the home page, and check the results.

Sure enough, the chart the the table should be displayed, but there is a one problem. The results contains all clients when we are interested in only one.

We can add filters to the report by `data-extra-params` to the `[data-report-widget]` element with the active client id and other parameters too..

```
{% extends 'ra/base_site.html' %}
{% load ra_admin_tags %}

{% block content %}
    {% get_report base_model='client' report_slug='clienttotalbalance' as client_
↪balances %}

    <div data-report-widget
         data-report-url="{{ client_balances.get_url }}"
         data-extra-params="&client_id={{ original.pk }}">

        <div data-report-chart height="50" data-report-default-chart="bar_chart"></
↪div>
        <div data-report-table></div>
```

(continues on next page)

(continued from previous page)

```
    </div>

{% endblock %}
```

Reload the page and you should see only the relevant data.

But the chart here is not very helpful, so we can remove it, also a table with only one row can be a little overkill as well, don't you think?

We can further enhance our widget by using the *data-success-callback data-success-callback* take a function name which will be called when server successfully replies with the report data. This javascript callback must accept two parameters

- response: The json response sent by the server and contains the results of the report (along with other data).
- $elem: the report jquery element *(ie the relevant '$('[data-report-widget]')')*

Let's see how would that look like

```
{% block content %}

{# Add this line #}
<h2>Balance is <span class="clientBalance"></span></h2>

{% get_report base_model='client' report_slug='clienttotalbalance' as client_balances
↪%}
<div data-report-widget
     data-report-url="{{ client_balances.get_url }}"
     data-extra-params="&client_id={{ original.pk }}"
     data-success-callback="displayBalance">
</div>
<div data-report-table></div>
{% endblock %}


{% block extra_js %}

    <script>
        function displayBalance(response, $elem) {
            $('.clientBalance').text(response['data'][0]['__balance__']);
        }
    </script>
{% endblock %}
```

So what did we do ?

1. we used *data-success-callback="displayBalance"* which should be accessible to the javascript context.
2. we accessed the response sent from the server *data* which is a list of the results, we accessed the first item in that array, and got the *__balance__* property

Hint: The default success callback *$.ra.report_loader.loadComponents* checks for the existence of elements with attr *[data-report-chart]* if found it calls *$.ra.report_loader..displayChart*. It also check for children elements with attr *[data-report-table]* , if found it calls *$.ra.datatable.buildAdnInitializeDatatable* and pass the response, $elem arguments.

Let's add another report.

```
{% get_report base_model='client' report_slug='productclientsales' as client_sales_of_
↪products %}
<div data-report-widget
     data-report-url="{{ client_sales_of_products.get_url }}"
     data-extra-params="&client_id={{ original.pk }}">

    <div data-report-table></div>
</div>
```

Now you should have a good idea on how you can use Ra framework to build your system.

1.3 Advanced topics

1.3.1 Settings

Ra have a wealth of settings that puts you in control.

Base Models Settings

See *Base Classes* for information on what are Ra Base Models and how they interact with the rest of the framework

`RA_BASEINFO_MODEL`

Default: `'ra.base.models.EntityModel'`

A dotted path to the Base Model from which all other model shall be inherited.

`RA_BASEMOVEMENTINFO_MODEL`

Default: `'ra.base.models.BaseMovementInfo'`

A dotted path to the Base Transaction Model from which all other transaction models shall be inherited.

`RA_QUANVALUEMOVEMENTITEM_MODEL`

Default: `'ra.base.models.QuantitativeTransactionItemModel'`

A dotted path to the Base Quantity / Value Transaction Model from which all other transaction models shall be inherited.

Flow Control Settings

`RA_ADMIN_SITE_CLASS`

Defaults `'ra.admin.admin.RaAdminSite'`

A dotted path to the main Ra Admin Site class. Make sure to inherit from `RaAdminSiteBase` in your custom admin site.

RA_ENABLE_ADMIN_DELETE_ALL

Default `False`

Control the availability of the admin action "Delete All" on all RaModelAdmin. While users

RA_FORMFIELD_FOR_DBFIELD_FUNC

Defaults `'ra.base.helpers.default_formfield_for_dbfield'`

A dotted path a universal hook that gets called on all 'formfield_for_db_field' on the framework. You can use this hook to universally control the widgets being displayed without needing to manually set it on each RaModelAdmin

The function should have this signature.

```
def default_formfield_for_dbfield(model_admin, db_field, form_field, request,␣
↪**kwargs):
    ...
```

RA_DEFAULT_FROM_DATETIME

Defaults to start of this year (ie first of January). Type *datetime*

RA_DEFAULT_TO_DATETIME

Defaults to start of the next year (ie First of January, current year + 1). Type *datetime*

RA_NAVIGATION_CLASS

Defaults to `'ra.utils.navigation.RaSuitMenu'`

A Dotted path to the navigation render menu.

This class is forked from Django suit

Customization

RA_THEME

Defaults to `'adminlte'`

If you want to create a new Ra theme, You can override the templates in another path and set it here .

RA_ADMIN_INDEX_TEMPLATE

Defaults to `'f'{RA_THEME}/index.html'`

RA_ADMIN_APP_INDEX_TEMPLATE

Defaults to `'f'{RA_THEME}//app_index.html'`

`RA_ADMIN_LOGIN_TEMPLATE`

Defaults to `f'{RA_THEME}/login.html'`

`RA_ADMIN_LOGGED_OUT_TEMPLATE`

Defaults to `f'{RA_THEME}/logged_out.html'`

`RA_ADMIN_SITE_TITLE`

Defaults to `_('Ra Framework')`

`RA_ADMIN_SITE_HEADER`

Defaults to `_('Ra Administration')`

`RA_ADMIN_INDEX_TITLE`

Defaults to `_('Statistics and Dashboard')`

Cache

`RA_CACHE_REPORTS`

Defaults to `True`

Enabling Caching for the Reports

`RA_CACHE_REPORTS_PER_USER`

Defaults to `True`

Enable Caching the report value not only per its parameters, but also per each user.

1.3.2 The Admin

Ra Site

Ra Site is a custom admin site. It provide you the theme and other goodies aimed to make the dashboard more usable.

ModelAdmin Classes

A subclass of admin.ModelAdmin with various different options

1. whole_changeform_validation
2. *View* page

`EntityAdmin` offer two important hooks to manage little bit complicated flow

1. it offer *EntityAdmin.pre_save(self, form, formsets, change)* It offers you a hook before saving the whole page to do any management you want. Like saving the total of the invoicelines in the Invoice.value field.
2. `whole_changeform_validation(self, request, form, formsets, change, **kwargs)()` Where you'll get a chance to validate the whole page forms and formsets

EntityAdmin

class `ra.admin.admin.`**`EntityAdmin`**(**args*, ***kwargs*)

`changelist_view`(*request*, *extra_context=None*)
The 'change list' admin view for this model.

`formfield_for_dbfield`(*db_field*, *request*, ***kwargs*)
Hook for specifying the form Field instance for a given database Field instance.

If kwargs are given, they're passed to the form Field's constructor.

`get_actions`(*request*)
Return a dictionary mapping the names of all actions for this ModelAdmin to a tuple of (callable, name, description) for each action.

`get_changelist`(*request*, ***kwargs*)
Return the ChangeList class for use on the changelist page.

`get_list_display`(*request*)
Return a sequence containing the fields to be displayed on the changelist.

`has_view_permission`(*request*, *obj=None*)
Return True if the given request has permission to view the given Django model instance. The default implementation doesn't examine the *obj* parameter.

If overridden by the user in subclasses, it should return True if the given request has permission to view the *obj* model instance. If *obj* is None, it should return True if the request has permission to view any object of the given type.

`post_save`(*request*, *new_object*, *form*, *formsets*, *change*)
Hook for doing any final logic after saving :param form: :param formsets: :param change: :return:

`pre_save`(*form*, *formsets*, *change*)
Hook for doing any final computation setting before saving :param form: :param formsets: :param change: :return:

`reversion_register`(*model*, ***kwargs*)
Registers the model with reversion.

`save_model`(*request*, *obj*, *form*, *change*)
Given a model instance save it to the database.

`whole_changeform_validation`(*request*, *form*, *formsets*, *change*, ***kwargs*)
A Hook for validating the whole changeform :param form: the ModelAdmin Form :param formsets: inline formsets :param kwargs: extra kwargs :return: True for valid [default] False for Invalid

TransactionAdmin

class `ra.admin.admin.`**`TransactionAdmin`**(**args*, ***kwargs*)

formfield_for_dbfield(*db_field*, *request*, ***kwargs*)
Hook for specifying the form Field instance for a given database Field instance.

If kwargs are given, they're passed to the form Field's constructor.

save_formset(*request*, *form*, *formset*, *change*)
Given an inline formset save it to the database.

TransactionItemAdmin

class ra.admin.admin.**TransactionItemAdmin**(*parent_model*, *admin_site*)

formfield_for_dbfield(*db_field*, *request*, ***kwargs*)
Hook for specifying the form Field instance for a given database Field instance.

If kwargs are given, they're passed to the form Field's constructor.

get_permission_override_model(*request*, ***kwargs*)
Return a string reprsentation of the model to look into its permissions, :param request: :param kwargs: :return:

get_ra_permission_codename(*action*, *model_name*)
Returns the codename of the permission for the specified action.

has_add_permission(*request*, *obj=None*)
Return True if the given request has permission to add an object. Can be overridden by the user in subclasses.

has_change_permission(*request*, *obj=None*)
Return True if the given request has permission to change the given Django model instance, the default implementation doesn't examine the *obj* parameter.

Can be overridden by the user in subclasses. In such case it should return True if the given request has permission to change the *obj* model instance. If *obj* is None, this should return True if the given request has permission to change *any* object of the given type.

has_delete_permission(*request*, *obj=None*)
Return True if the given request has permission to change the given Django model instance, the default implementation doesn't examine the *obj* parameter.

Can be overridden by the user in subclasses. In such case it should return True if the given request has permission to delete the *obj* model instance. If *obj* is None, this should return True if the given request has permission to delete *any* object of the given type.

view_on_site = False
To simplify complex forms with inline , making inline permission reflect base form permission permission_override_model can be True, False , str Or ModelBase

1.3.3 Reporting

As you may have seen in the tutorial Part 2 *Reporting and Charting* , you can create a report by creating a subclass of `ReportView` class and register it with the decorator `@register_report_view`

Let's look at a bare minimum report

```
@register_report_view
class SimpleReport(ReportView):
    base_model = Product
    report_model = SalesLineTransaction
    report_title = 'Simple report'
    columns = ['slug', 'product__name', 'quantity', 'price', 'value']
```

The model in which this report must be imported during django load. Preferably on AppConfig.ready()

By registering this report, the dashboard have now a new menu item "reports", with a Product sub menu in which you'll find this report FilterForm and results.

Report View

A view class which represent a report with a default structure..

_ document hooks , cache _

Report Form

The report form get generated automatically and you can customize it on several levels. By default the filter form contains

1. A Date to filter
2. All foreign keys found in the `report_model` , displayed as a SelectMultiple Widget.
3. In case of a cross tab report, it shows a check "Show the rest".

The report form is responsible for delivering those filters into a queryset filters and hand them to the ReportGenerator

Report JSON Response Structure

// todo

Javascript

Report Loader API

Coming soon

1.3.4 Base Classes

EntityModel

class ra.base.models.**EntityModel**(**args*, ***kwargs*)
The Main base for Ra *static* models Example: Client , Expense etc..

classmethod get_class_name()
return the class name, usable when a ra model is mimicking (ie:proxying) another model. This method is used is get_doc_type_* functions, This method is made to avoid to repeat registered doc_type to make adjustments

classmethod get_doc_type_neuter_list()
Returns List of Identified doctype that have a neuttral effect on the entity

classmethod get_model_name()
A convenience method to get the base model name, needed in templates :return:

get_next_slug(*suggestion=None*)
Get the next slug If it's a new instance and the slug is not provided, we try and attempt a serial over the already added slugs in relation to the model :return:

get_pk_name()
This is used to get the full name of the primary key, a bit hackish but is important for reports. :return:

classmethod get_redirect_url_prefix()
Get the url for the change list of this model :return: a string url

classmethod get_report_list_url()
Return the url for the report list for this model :return: a string url

get_title()
A helper function to get a custom title of the instance if needed :return:

classmethod get_verbose_name_plural()
A convenience method to get the base model verbose name, needed in templates :return:

save(*force_insert=False*, *force_update=False*, *using=None*, *update_fields=None*)
Save the current instance. Override this in a subclass if you want to control the saving process.

The 'force_insert' and 'force_update' parameters can be used to insist that the "save" must be an SQL insert or update (or equivalent for non-SQL backends), respectively. Normally, they should not be set.

TransactionModel

class ra.base.models.**TransactionModel**(**args*, ***kwargs*)

classmethod get_doc_type()
Return the doc_type :return:

save(*force_insert=False*, *force_update=False*, *using=None*, *update_fields=None*)
Custom save, it assign the user As owner and the last modifed it sets the doc_type make sure that dlc_date has correct timezone ?! :param force_insert: :param force_update: :param using: :param update_fields: :return:

title
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

TransactionItemModel

class ra.base.models.**TransactionModel**(**args*, ***kwargs*)

classmethod get_doc_type()
Return the doc_type :return:

save(*force_insert=False*, *force_update=False*, *using=None*, *update_fields=None*)
Custom save, it assign the user As owner and the last modifed it sets the doc_type make sure that dlc_date

has correct timezone ?! :param force_insert: :param force_update: :param using: :param update_fields: :return:

title
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

QuantitativeTransactionItemModel

class `ra.base.models.`**TransactionModel**(**args*, ***kwargs*)

classmethod get_doc_type()
Return the doc_type :return:

save(*force_insert=False*, *force_update=False*, *using=None*, *update_fields=None*)
Custom save, it assign the user As owner and the last modifed it sets the doc_type make sure that dlc_date has correct timezone ?! :param force_insert: :param force_update: :param using: :param update_fields: :return:

title
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

1.3.5 Customizing Print

Not all what is displayed on the screen can (or should be) printed. Learn how to customize how reports are printed

1.3.6 Javascript

Work in progress

Ra comes with a wealth of javascript code utilities and 3rd parties. Learn more about tools available at your disposle here

Public function

Widget Report loader

1.3.7 Permissions

1.3.8 Theming

Dashboard Theme

Dashboard theme is mainly powered by Django Jazzmin , built on top of AdminLTE

Check out Django Jazzmin documentation

Theming Charts colors

Documentation In progress

1.3.9 Road Map

- Bring back top search
- Bring back print on page
- Bring back activity log
- crosstab charting capabilities

Documentation

- Document Settings

Tests:

- Test crosstab on doc_type
- test timeseries on doctype

1.4 Release Notes

1.4.1 V1.0.0.0

Public Stable Release

- Reporting Engine is moved to its own new package [Slick Reporting](https://github.com/ra-systems/django-slick-reporting).
- Framework no longer depends Postgres Database specifically.
- Dropped in house AdminLTE implementation in favor of *django-jazzmin*
- Re-organize documentation
- Adds Highcharts support and making it the default.

1.4.2 v0.1.1 (and earlier)

1.5 FAQ

1.5.1 What's the meaning of "Ra" ?

Ra is the name of the deity of the sun in the ancient Egyptian mythology.

From Wikipedia : By the Fifth Dynasty in the 25th and 24th centuries BC, he (Ra) had become one of the most important gods in ancient Egyptian religion, identified primarily with the noon sun. Ra was believed to rule in all parts of the created world: the sky, the Earth, and the underworld.

1.5.2 What Ra framework is not?

Ra is not a ready made ERP solution, rather, a framework that provide you tools to build one effortlessly and in style. To check how easy it is, go through the tutorial. By the end of it, you'd have a working sales monitor app with a dozen reports and charts in couple of hours.

1.5.3 How this project got started ?

The author (@RamezIssac) always worked in the field of ERP/accounting solutions since in 2002. This framework was triggered by author work to supply an ERP solution to a big holding company in Python and Django.

Then the framework got loaded with ERP different modules (sales / purchases / payment / General ledger etc. . .) and it's not till November 2019 that was decided to open source the core.

After that decision was made, the author went on and removed all "specific" modules which contained many assumptions, simplified the api, substitute paid libraries with open source ones, write a tutorial / documentation and . . . . here we are :)

In April 2020, the Ra reporting engine was released as a standalone package. Django Slick Reporting

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index